Haaga-Helia
ammattikorkeakoulu Oy

Oppiva tekoäly peliohjelmoinnissa

Daniel Laitinen

12.12.2016

Haaga-Helia
ammattikorkeakoulu Oy

Tiivistelmä

Tekijä(t)
Daniel Laitinen

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Raportin/Opinnäytetyön nimi
Oppiva tekoäly peliohjelmoinnissa

Sivu- ja liitesivumäärä
22 + 0

Tämän opinnäytetyön tarkoituksena on toteuttaa oppivaa tekoälyä hyödyntävä mobiilipeli Android-alustalle. Tavoitteenani on selvittää, miten oppiva tekoäly toteutetaan mobiilipeliin ja arvioida sen tuomia hyötyjä sekä mahdollisia haittoja.

Teoriaosuudessa tarkastellaan alkuun tekoälyn määritelmää ja sen jakamista heikkoihin sekä vahvoihin tekoälyihin. Seuraavaksi käydään läpi pelitekoälyjen erilaisia rooleja peleissä ja tutustaan tarkemmin opinnäytetyön aiheena olevaan oppivaan tekoälyyn.

Suunnitteluosiossa määritellään valitut tekniikat, joilla toteutan tekoälyn peliin. Erilaisiin toteutustapoihin tutustumisen jälkeen opetustavaksi valikoitui vahvistettu oppiminen käyttämällä Q-oppimisen algoritmia sekä neuroverkkoa.

Pelin toteutuksesta käydään alkuun läpi yleisesti pelin idea, tavoite ja suunnitteluvaiheet. Käytettäväksi pelimoottoriksi valitsin Unityn. Peliradan toteuttamisen jälkeen peliin lisätään oppiva tekoäly ja käydään läpi lisäämisen tuomia haasteita. Valitut toteutustavat käydään läpi ja perustellaan. Lopuksi arvioidaan peliprojektin onnistumista ja käydään läpi mahdollisia ongelmakohtia sekä kehityskohteita.

Yhteenvedossa arvioin koko projektin onnistumista sekä tavoitteisiin pääsemistä. Lisäksi vastaan tutkimuskysymyksiin ja arvioin omaa oppimistani tämän projektin aikana.

Asiasanat
Oppiva tekoäly, Unity, C#, Android

Sisällys

1 Johdanto

Alkuun pelit olivat lähinnä kaksinpelejä, eikä niissä yleensä ollut minkäänlaista tekoälyä. Tämä on muuttunut ajan kuluessa ja nykyään tekoäly on suuri osa modernia pelikehitystä. Tekoälyn laadusta riippuen se voi joko parantaa peliä huomattavasti tai pilata sen. (Giantbomb.) Kaikki tähän mennessä luodut tekoälyt ovat niin kutsuttuja heikkoja tekoälyjä, joiden tavoitteena on vaikuttaa älykkäältä. Useimmat tekoälyt koostuvat yksinkertaisesti ehtolauseista, joiden avulla tekoäly reagoi eri tilanteisiin. (Kahanda, I. 2011.)

Pelinkehitys, varsinkin mobiilipuolella, on ollut suuressa kasvussa lähivuosina, joten siihen liittyvien aiheiden tutkimus on ajankohtaista. Mobiilipuolelle ei ole julkaistu juurikaan pelejä, jotka hyödyntävät oppivaa tekoälyä. Tämän opinnäytetyön tavoitteena on selvittää, että miten oppiva tekoäly lisätään mobiilipeliin ja arvioida sen tuomia hyötyjä. Tarkoituksena on myös pohtia mahdollisia ongelmia ja haasteita, mitä oppiva tekoäly tuo mukanaan.

Kiinnostuin peliohjelmoinnista Haaga-Helian opiskeluideni aikana ja aloin omatoimisesti opiskelemaan aiheesta enemmän. Melko pian vastaan tulikin tarve toteuttaa yksinkertainen tekoäly yksinpeliä varten. Tästä alkoi kiinnostukseni pelitekoälyjä kohtaan. Koska peliohjelmoinnista ja yksinkertaisen tekoälyn toteuttamisesta on aikaisemminkin tehty opinnäytetöitä, päätin laajentaa aihepiiriä oppivaan tekoälyyn. Alun perin suunnittelin tekoälyn opettamista pelaamisen aikana, samalla kun tekoäly pelaa pelaajaa vastaan. Hyvin pian työn aloitettuani selvisi, ettei tekoälyn opettamista yleensä toteuteta pelaamisen aikana. Useimmiten opettaminen tehdään pelistudiolla, eikä oppimista enää tapahdu valmiissa pelissä. Tästä johtuen päädyin hieman erilaiseen ratkaisuun, jossa vaikeusastetta voi nostaa antamalla tekoälyn opetella pelaamaan peliä itse. Tekoäly käytännössä opettelee pelaamaan pelaamisen aikana, mutta tässä tapauksessa pelaaja ei ole mukana vaikuttamassa oppimiseen. Tämän opinnäytetyön pääpaino on oppivan tekoälyn luomisessa Android-peliin, joten itse pelin muita tekovaiheita ei kuvata tarkasti ja ulkoasu jää tarkoituksella melko yksinkertaiseksi. Laajempi suorituskykytestaus eri matkapuhelimilla ei kuulu työn aihepiiriin, mutta toimivuus Android-käyttöjärjestelmällä testataan yhdellä matkapuhelimella.

Teoriaosuudessa käydään alkuun läpi yleisesti tekoälyn määritelmää. Kyseisessä luvussa tutustutaan erityisesti tekoälyn rooleihin peleissä, sekä oppivaan tekoälyyn, joka on oleellisin asia tämän opinnäytetyön kannalta. Seuraavaksi suunnitteluosuudessa käydään läpi keskeiset peliprojektin toteutukseen liittyvät aiheet. Tärkein näistä on Q-oppiminen ja projektin edetessä mukaan tulivat myös neuroverkot. Näiden jälkeen käydään läpi projektin toteutus ideasta valmiiksi peliksi. Lopuksi arvioidaan projektin onnistumista sekä vasta-

taan esitettyihin tutkimuskysymyksiin.

1.1 Käsitteet

2D peli — Peli, jossa voi liikkua vain kahdessa ulottuvuudessa, esim. vain pysty- ja vaakasuunnassa.

3D peli — Peli, jossa voi liikkua kolmessa ulottuvuudessa (pysty-, vaaka- ja syvyyssuunnassa).

AI — Tarkoittaa tekoälyä. Tulee englannin kielen sanoista Artificial Intelligence.

BFS — Tulee sanoista Breadth First Search. Algoritmi, joka käy dataa läpi leveyssuunnassa.

C# — Microsoftin kehittämä C Sharp ohjelmointikieli.

DFS — Tulee sanoista Depth First Search. Algoritmi, joka käy dataa läpi syvyyssuunnassa.

Komentosarja — Sarja ohjelmoituja komentoja, joita ohjelma/tekoäly noudattaa.

NPC — Tekoälyn ohjaama pelihahmo. Tulee englannin kielen sanoista Non-Player Character.

Q-oppiminen — Q-learning, vahvistetun oppimisen algoritmi

Törmäyksen tunnistin — Collider Unityssa. Unityn oma komponentti.

Unity — Unity Technologiesin kehittämä pelimoottori, jota käytetään pelien kehittämiseen useille eri alustoille.

2 Tekoäly

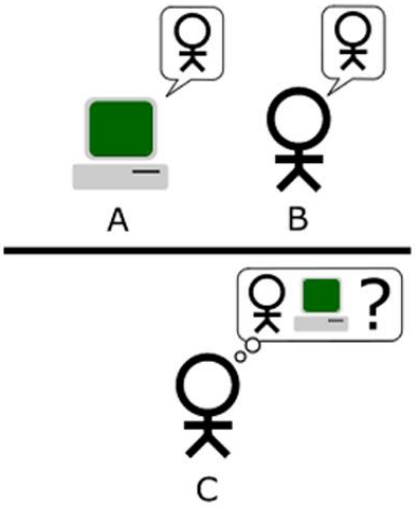
Tekoäly on tietojenkäsittelytieteen alue, jossa tavoitteena on kehittää laitteita, jotka matkivat ihmistä ja osaavat suorittaa tehtäviä kuten ihminen. Tietotekniikan saapuminen ja nopea kehittyminen viimeisen 50 vuoden aikana on auttanut tutkijoita pääsemään lähemmäs tätä päämäärää. Silti kaikki tähän mennessä luodut tekoälyt ovat vielä kaukana ihmisen älykkyydestä. Tekoälyt jaetaan yleisesti kahteen luokkaan: heikko tekoäly ja vahva tekoäly. Kaikki tähän mennessä luodut tekoälyt kuuluvat heikkoihin tekoälyihin. (Kahanda, I. 2011.) Tässä luvussa käydään läpi heikon ja vahvan tekoälyn määritelmät, tutustutaan tekoälyn rooleihin peleissä, sekä syvennytään tämän opinnäytetyön aiheena olevaan oppivaan tekoälyyn.

2.1 Heikko ja vahva tekoäly

Heikon tekoälyn perustana on yksinkertaisesti, että laitteet voidaan saada käyttäytymään, aivan kuin ne olisivat älykkäitä. Shakkiohjelmaa vastaan pelatessa voi tuntua, että tekoäly tekee tietoisia siirtoja. Oikeasti kaikki siirrot perustuvat tekoälyyn syötettyihin käskyihin, joita se noudattaa aina tietyissä tilanteissa. (Kahanda, I. 2011.) Tekoäly ei oikeasti tiedä shakista mitään, eikä se osaa itse arvioida, mikä on hyvä ja mikä on huono siirto. Se vain analysoi tilanteen sille ohjelmoidun logiikan mukaisesti ja tekee siirron sen perusteella. (Hietala, J. 2012).

Heikko tekoäly eroaa vahvasta siinä, että heikko osaa suorittaa vain ennalta määrättyjä tehtäviä perustuen sille annettuihin ohjeisiin. Näiden ohjeiden avulla se soveltaa parasta ratkaisua kyseiseen tilanteeseen. Vahva tekoäly taas kykenee itsenäiseen ajatteluun kuten ihminen, mutta tällaista tekoälyä ei ole vielä onnistuttu luomaan. (Kahanda, I. 2011.) Vahva tekoäly osaisi järkeillä ja ajatella sekä tehdä kaikkea muuta mitä ihminenkin. Yleinen mielipide on, ettei tällaista tekoälyä saada ikinä luotua, tai vaikka saataisiinkin, siihen menee vielä hyvin kauan. Tulevaisuudessa keinotekoiset neuroverkot voivat olla mahdollinen tapa toteuttaa vahva tekoäly. (Kahanda, I. 2011.)

Yksi ehdotettu tapa vahvan tekoälyn erottamiseen heikosta on Turingin testi. Kuviossa 1 näkyy kyseisen testin menetelmä. Tietokone A ja henkilö B ovat erillisissä huoneissa. Henkilö C keskustelee molempien kanssa kolmannesta huoneesta tietokoneen välityksellä ja yrittää päätellä, onko A vai B ihminen. (Bilby. 2008.) Mikäli tietokonetta ei erota ihmisestä, kyseessä on vahva tekoäly (Copeland, J. 2000).



Kuvio 1. Turingin testi (Bilby 2008).

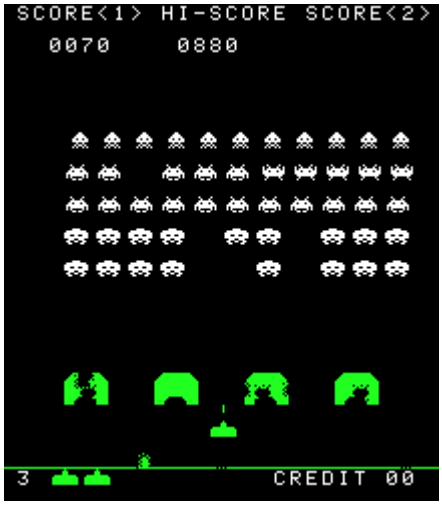
Turingin testin toimivuus on kyseenalaistettu usein. Yksi väite on, että on mahdollista suunnitella heikko tekoäly, joka pääsisi testistä läpi. Tätä varten tarvittaisiin vain jättimäinen taulu, jossa olisi oikeat vastaukset kaikkiin Turingin testin kysymyksiin. Tämän perusteella tekoäly, joka ei täytä älykkyyden määritelmää, pystyisi läpäisemään testin. (Copeland, J. 2000.) Myös Stuart Russel, Kalifornian yliopiston ”Center for Intelligent Systems”in perustaja, totesi hiljattain, että melkein kukaan tekoälyn tutkija ei tähtää Turingin testin läpäisemiseen, paitsi ehkä harrastusmielessä. Hänen mukaansa Turingin testiä ei muutenkaan kehitetty tekoälyjen tavoitteeksi, vaan todistamaan skeptisille ihmisille, että älykkäät koneet eivät tarvitse tietoisuutta käyttäytyäkseen älykkäästi. (Prado, G. 2015.)

LaCurtsin mukaan suurin osa Turingin testin toimivuuden kyseenalaistavista väitteistä on vääriä tai epäoleellisia. Hänen mukaansa väite, jossa heikko tekoäly läpäisee Turingin testin käyttämällä suurta taulua, ei toimisi kielten kohdalla. Jotta tekoäly osaa kommunikoida järkevästi kiinaksi, sen täytyy ensin oppia kieli. LaCurts myöntää, että tulevaisuudessa Turingin testiä voidaan joutua muokkaamaan, kun ymmärrämme älykkyydestä enemmän. Hänen mukaansa tällä hetkellä yksi parhaista tavoista oppia älykkyydestä, on luoda älykkäitä koneita, joilla on päämääränä läpäistä Turingin testi. (LaCurts, K.)

2.2 Tekoäly peleissä

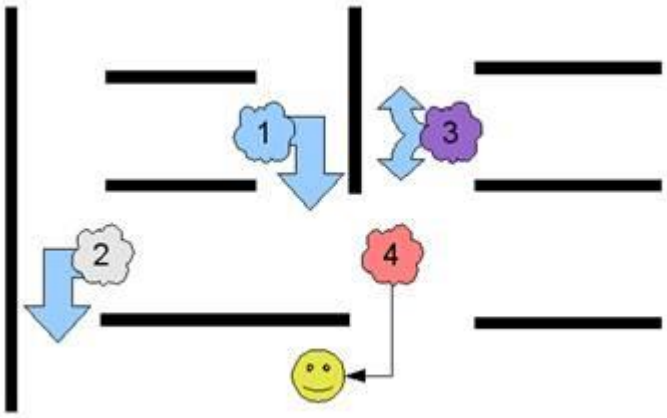
Pelitekoälyistä puhuttaessa selkein nähtävä esimerkki on tekoälyn ohjaamat hahmot, NPC:t (Non-Player Characters). Kehoen (2009) mukaan tekoäly hoitaa myös ennalta määriteltyjä tapahtumia, esimerkiksi jos rakennus romahtaa pelissä, tekoäly on siitä vastuussa.

Alun perin pelit olivat kaksinpelejä, joissa ei tarvittu tekoälyä, mutta tämä muuttui Space Invaders -pelin myötä (kuvio 2). Siinä viholliset liikkuivat ryhmässä ja ampuivat sattumanvaraisesti. Space Invadersin jälkeen tekoäly on kehittynyt huomattavasti. Seuraavaksi tulivat monimutkaisemmin liikkuvat viholliset ja sattumanvaraiset tapahtumat, kuten yhden vihollisen erkaantuminen ryhmästä. (Giantbomb.)



Kuvio 2. Space Invaders (Pearce, A. 2016).

Myös Pacmanissa neljällä haamulla oli jokaisella erilainen ”persoonallisuus” ja käyttäytyminen. Kuviossa 3 on esitetty haamujen liikkumislogiikka. Yksi haamu kääntyy aina oikealle, toinen vasemmalle, yksi sattumanvaraisesti vasemmalle tai oikealle ja viimeinen pelaajaa kohti. Yksittäistä haamua vastaan pelaaminen olisi helppoa ja sen toimintalogiikan pystyisi päättelemään helposti, mutta ryhmässä niiden liikkumislogiikka vaikuttaa monimutkaisemmalta ja jopa suunnitelmalliselta. Oikeasti vain yksi haamu edes tarkistaa pelaajan sijainnin, loput kolme eivät tiedä pelaajan sijainnista oikeasti mitään. (Kehoe, D. 2009.)



Kuvio 3. Haamujen käyttäytyminen Pacmanissa (Kehoe, D. 2009).

Nykyään moderneissa peleissä viholliset osaavat jo liikkua ryhmässä, kommunikoida keskenään, kiertää pelaajan taakse ja hyökätä sivusta. Pelien tekoälyjen logiikka vaihtelee tarkoituksen ja tehtävän mukaan. (Giantbomb.) Seuraavissa kappaleissa on mainittu muutamia peleissä esiintyviä tekoälyjen tehtäviä ja toimintalogiikoita.

2.3 Yksinkertaiset pelitekoälyt

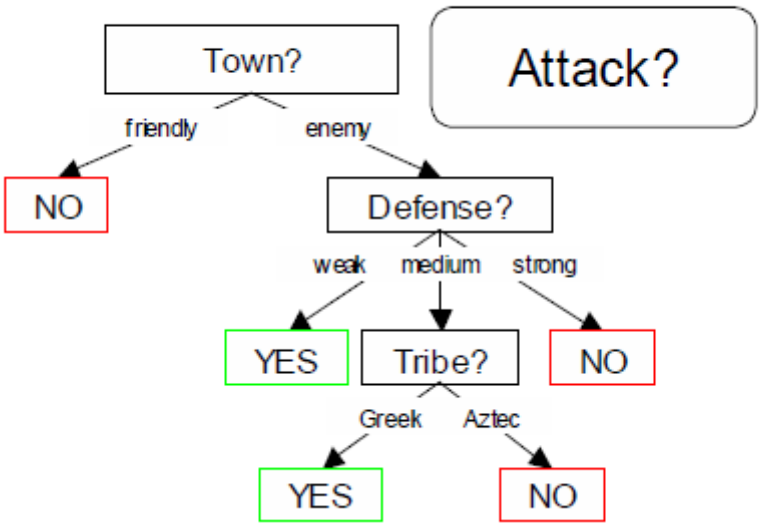
NPC:t käyttävät yleensä paljon oppivaa tekoälyä yksinkertaisempaa ratkaisua. Yleisin ja yksinkertaisin peleissä käytetty tekoäly koostuu erilaisista ehdoista. Tekoäly reagoi tilanteisiin annettujen ehtojen mukaisesti. Ehtona voisi olla esimerkiksi: jos vihollinen menee seinän taakse, heitä kranaatti. Peliin saadaan hieman enemmän vaihtuvuutta, jos komentosarjoihin lisätään sattumanvaraisuutta. Jos vihollinen menee seinän taakse, heitä kranaatti tai juokse perään. Molemmille vaihtoehdoille on määritelty tietyn suuruinen mahdollisuus. NPC:iden käyttäytymistä voidaan muokata hahmosta riippuen. Jos pelissä NPC on vaikka poliisi, hän tuskin heittäisi kranaattia rosvon perään. Hän voisi esimeriksi jäädä suojaan ja kutsua apujoukkoja, kun taas rynnäkköpoliisi voisi mennä pelaajan perään. Näillä vaihtoehdoilla saadaan pelihahmoille luonnetta ja pelille vaihtelevuutta. (Giantbomb.)

Varsinkin kilpailullisissa peleissä käytetään usein tekoälyä, jota kutsutaan kuminauhaksi. Sen ideana on lisätä haastetta nopeuttamalla takana tulevia NPC-kuskeja, jotta ne saavat pelaajan kiinni. Tämä toteutetaan lisäämällä kyseisten NPC:iden autojen nopeutta ja parantamalla niiden ohjausta, kunnes ne saavuttavat pelaajan. (Giantbomb.)

2.4 Oppiva tekoäly

Oppivalla tekoälyllä on käytännössä potentiaalia mukautua pelaajan pelityyliin ja tarjota mukautuvaa haastetta. Tällä tavalla voidaan myös luoda uskottavampia hahmoja, jotka reagoivat ympäristöönsä aidommin. Oppivuus voi helpottaa pelikohtaisten tekoälyjen luomista: tekoäly voi itse oppia peliympäristöstä. Opettamistekniikoita on useita erilaisia. Tekoälyn opettaminen voi olla kaikkea muutamien numeroiden muuttamisesta monimutkaisten neuroverkkojen päivityksiin. Oppivaa tekoälyä käytetään useimmiten tekoälyn opettamiseen pelin kehittämisen aikana pelistudiolla. Ennen julkaisua tekoäly jätetään tiettyyn tilaan, josta se ei enää kehity. Jos tekoäly yrittäisi mukautua pelaajan pelityyliin, tämä saattaisi aiheuttaa ennalta-arvaamattomia ongelmia. Oppiminen vaatii suuria määriä toistoja tapahtuakseen järkevästi, joten pelaajan pelityyliin mukautuminen olisi vaikeaa. (Millington, I. 2006.) Oppivaa tekoälyä käytetään muutamissa peleissä, joista yhtenä tunnettuna esimerkkinä on Lionhead Studiosin Black & White. Siinä pelaaja on jumala ja ihmiset suhtautuvat pelaajaan hänen tekojensa mukaan (Microsoft b, 12). Pelissä on myös mah-

dollista kouluttaa lemmikkiä (eng. creature). Kuviossa 4 on lemmikin käyttämä logiikkapuu sen saatua käsky hyökätä kylään. Logiikkapuun rakenne perustuu lemmikin aikaisempiin kokemuksiin pelissä. Lemmikki myös seuraa pelaajan tekoja ja osaa muuttaa käytöstään sen mukaan miellyttääkseen omistajaansa, eli pelaajaa. (Wexler, J. 2002.)



Kuvio 4. Lemmikin logiikkapuu hyökkäyskäskyn saatuaan (Wexler, J. 2002).

Oppivaa tekoälyä on käytetty pelin tekoälyn lisäksi myös pelaajana. Google kehitti vuoden 2015 alussa ensimmäisen tekoälyn, joka kykenee opettelemaan erilaisia asioita. Tekoäly on nimeltään ”Deep Q-Network”, lyhyesti DQN. Se tarvitsee vain normaalin pöytäkoneen verran prosessointitehoja. DQN:n ideana oli opetella pelaamaan Atari 2600 -konsolin pelejä ja kehittyä paremmaksi niissä. Tutkijat testasivat sitä 49:llä Atari 2600 pelillä, kuten Pong ja Space Invaders. Ainoa syöte mitä DQN sai peleistä, olivat ruudun pikselit ja pistetilanne. Käytännössä DQN opetteli pelaamista sattumanvaraisilla näppäinpainalluksilla ja tutki, miten se vaikuttaa pisteisiin. Muutaman viikon päästä DQN toimi jo ammattilaispelaajien tasolla monissa peleissä, jotka vaihtelivat 2D ammuntapeleistä 3D ajopeleihin. DQN sai 75% ihmisen pisteistä yli puolissa peleistä. DQN kykenee ”muistamaan” aikaisempia kokemuksia ja käyttää niitä hyväksi päätöksien teossa. Aivan ihmismäisesti DQN ei kuitenkaan käyttäydy, sillä ihmisaivot eivät muista asioita samalla tavalla kuin se. Yleensä ihmisen aivot muistavat paremmin tilanteet, joissa on ollut tunnelatausta ja ne vaikuttavat päätöksiin enemmän. Tätä DQN ei vielä kykene tekemään, mutta tutkijoiden mukaan se pyritään ottamaan huomioon seuraavissa versioissa. (Lewis, T. 2015.)

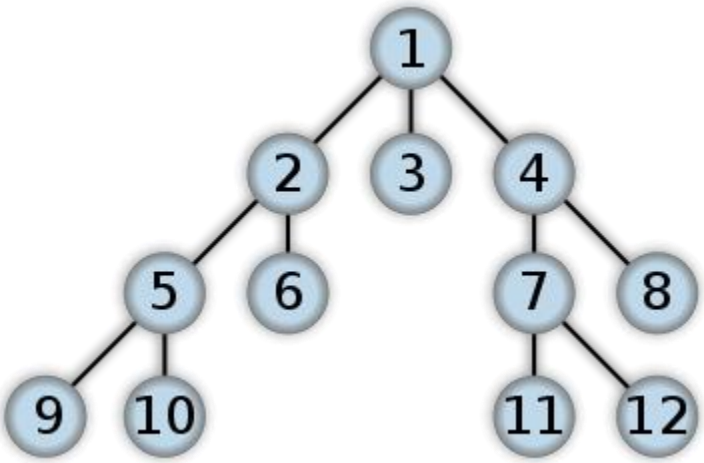
Hiljattain myös Lontoossa opiskeleva Matthew Lai onnistui luomaan tekoälyn nimeltään Giraffe, joka opetteli itse pelaamaan shakkia. Giraffe harjoitteli kolme päivää itseään vastaan, jonka aikana se kehittyi ”International Master” -tasolle, eli shakkipelaajien parhaan

2.2% joukkoon. Giraffe analysoi siirron kolmessa osassa. Ensin se tarkistaa, kenen vuoro on tehdä siirto ja mitä shakkinappuloita on saatavilla. Sitten se arvioi pelitilanteen nappuloiden paikkojen mukaan. Viimeiseksi se arvioi, mitkä siirrot ovat mahdollisia ja suhteuttaa ne miljooniin aikaisemmin tekemiinsä skenaarioihin. Toisin kuin aikaisemmat shakkitekoälyt, Giraffe oppii mitkä siirrot toimivat ja mitkä eivät. (Lee, R. 2015.)

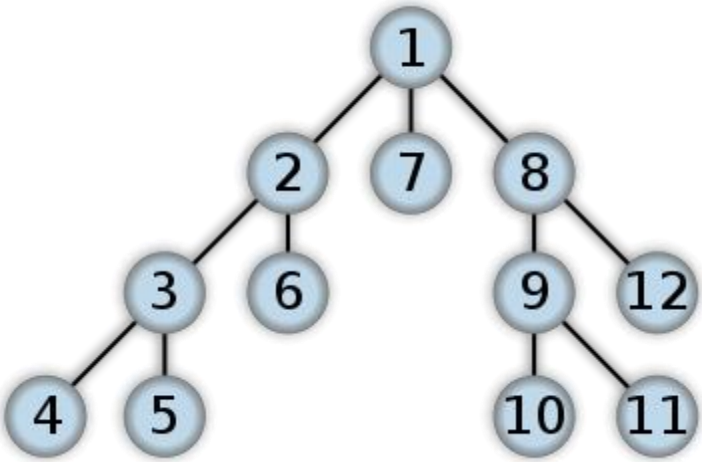
2.5 Reittihaku

Reittihakua tarvitsee lähes kaikissa peleissä, joissa on tekoälyn ohjaamia hahmoja. Reittihaun tehtävänä on yksinkertaisesti siirtää NPC, tai mikä tahansa muu peliobjekti, pisteestä A pisteeseen B. Siirron tulisi tietysti tapahtua lyhintä ja järkevintä reittiä pitkin. 3D peleissä maastonmuodot aiheuttavat helposti ongelmia ja huomioon täytyy myös ottaa pelaajan sijainti ja käyttäytyminen (esim. ampuminen). (Giantbomb.)

Kaksi yksinkertaista reittihakualgoritmia ovat BFS (Breadth First Search) ja DFS (Depth First Search). Kumpikaan ei ota huomioon pisteiden välisiä etäisyyksiä, eikä muutenkaan etsi lyhintä reittiä. Molemmat käyvät verkkoa läpi systemaattisesti, mutta eri tavalla. BFS (Kuvio 5) käy verkkoa läpi kerros kerrallaan, kun taas DFS (Kuvio 6) käy verkkoa läpi syvyyssuunnassa. Molemmille algoritmeille on omat käyttökohteensa riippuen tilanteesta, mutta pohjimmiltaan molemmat ovat hyvin yksinkertaisia. (Hietala, J. 2012.)

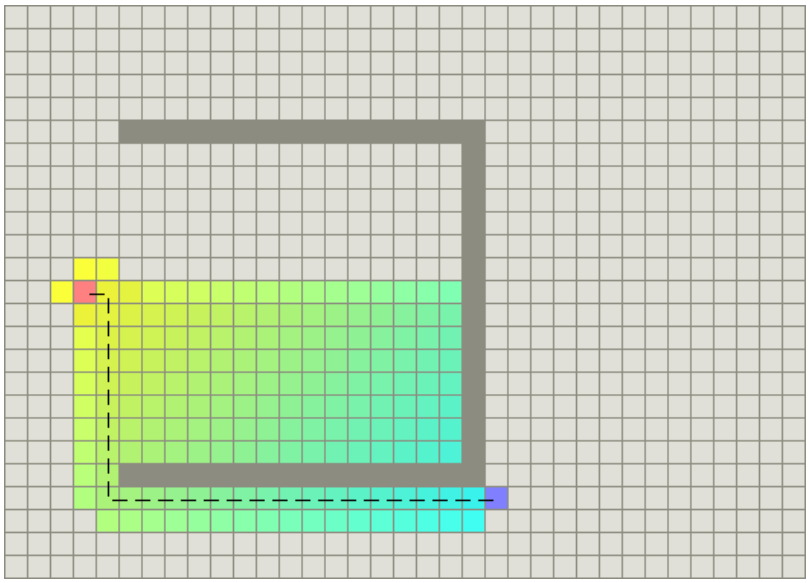


Kuvio 5. BFS hakujärjestys, numerot esittävät siirtymäjärjestystä solmuissa (Programmer Interview).



Kuvio 6. DFS hakujärjestys, numerot esittävät siirtymäjärjestystä solmuissa (Programmer Interview).

BFS:ää ja DFS:ää huomattavasti kehittyneempi, monimutkaisempi ja tällä hetkellä suosituin reittihakualgoritmi on nimeltään A* (A-tähti), jota on käytetty mm. reaaliaikastragiapeli Warcraft III:ssa. A* etsii lyhimmän reitin kahden pisteen välillä. Se pohjautuu Dijkstran ja Greedy Best-First -algoritmeihin. (Eranki, R. 2002.) Dijkstran algoritmi aloittaa haun lähtöpisteestä ja sen taataan löytävän lyhin reitti lähtöpisteestä päämäärään. Best-First -algoritmi toimii samankaltaisella tavalla, mutta se aloittaa haun päämäärästä, eikä sen taata löytävän lyhintä reittiä. Se on kumminkin paljon Dijkstran algoritmia nopeampi. A* käyttää heuristiikkaa itsensä ohjaamiseen, on yhtä tarkka kuin Dijkstran algoritmi ja yhtä nopea kuin Best-First algoritmi. (Patel, A.)

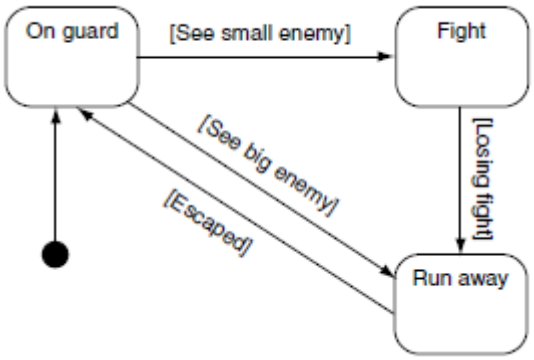


Kuvio 7. A* algoritmin reitinetsintä (Patel, A).

Kuviossa 7 on esimerkki A* algoritmin reitinetsinnästä. Se yhdistää Dijkstran algoritmin käyttämät tiedot (joka suosii aloituspaikkaa lähellä olevia pisteitä) Best-First algoritmin käyttämien tietojen kanssa (joka suosii lähellä päämäärää olevia pisteitä). A* laskee jokaiselle ruudukon pisteelle arvon, joka perustuu molempiin tietoihin ja päättää liikkumisesta sen mukaan. Kuviossa 7 näkyvä reitti on sama kuin Dijkstran algoritmin laskema reitti, mutta A* laski sen huomattavasti nopeammin. (Patel, A.)

2.6 Tilakone

Tilakoneita (state machine) käytetään usein komentosarjojen kanssa peleissä. Tilakone tarkoittaa, että tekoälyhahmolla on erilaisia tiloja, joiden välillä se voi siirtyä ennalta määrättyjen sääntöjen mukaisesti. Kuviossa 8 esitetyn tilakoneen mukaan tekoäly ei voi paeta, ennen kuin se on nähnyt suuren vihollisen. Tekoäly on aina kerrallaan vain yhdessä tilassa. Kuviossa 8 tekoälyn oletustila on vahdissa oleminen, kunnes se havaitsee vihollisen. Tekoäly hyökkää, jos vihollinen on pieni. Tekoäly pakenee, jos vihollinen on suuri tai jos se on häviämässä taistelun. Pelitekoälyjen kohdalla tällaista rakennetta kutsutaan yleensä äärelliseksi tilakoneeksi (finite state machine). Tilakone ottaa huomioon niin ympäröivän pelimaailman kuin hahmon sisäisen tilankin (esim. elämäpisteet). (Millington, I. 2006. 318-320.)



Kuvio 8. Yksinkertainen tilakone (Millington, I. 2006. 319).

3 Oppivan tekoälyn suunnittelu

Oppiva tekoäly on suosittu aihe peleissä, mutta sitä ei kuitenkaan käytetä peleissä niin paljoa kuin voisi luulla. Oppivuuden lisääminen peliin vaatii tarkkaa suunnittelua ja riskien tuntemista. Joskus oppivan tekoälyn mainonta on todellisuutta hienompaa. Eri opetustekniikoiden ongelmien tunteminen ja niiden realistinen käyttäminen mahdollistavat oppivan tekoälyn onnistuneen käyttämisen peleissä. Opetustekniikoita on useita erilaisia ja pelien kohdalla suosittu aihe on vahvistettu oppiminen. (Millington, I. 2006. 563-565, 612.) Tässä luvussa käydään läpi toteutettavaa peliä varten valitut tekniikat. Näihin lukeutuu vahvistettu oppiminen, jonka toteuttamiseen valitsin Q-oppimisen algoritmin, sekä algoritmia varten mukaan otettu neuroverkko.

3.1 Vahvistettu oppiminen

Vahvistettu oppiminen (reinforcement learning) on suosittu aihe pelitekoälyjen yhteydessä. Tällä nimellä kutsutaan erilaisia tekniikoita jotka perustuvat kokemuksesta oppimiseen. Yleisimmässä muodossaan se sisältää kolme komponenttia: kokeilustrategia pelin eri toimintojen kokeiluun, vahvistettu funktio joka antaa palautetta toimintojen toimivuudesta/hyvyydestä ja oppimissääntö, joka linkittää nämä kaksi yhteen. Kaikista elementeistä on erilaisia toteutuksia ja optimointeja riippuen ohjelmasta. (Millington, I. 2006. 612-613.)

Russelin ja Norvigin (2009, 831) mukaan vahvistetun oppimisen voi kiteyttää seuraavalla tavalla: ajattele pelaavasi peliä, jonka sääntöjä et tiedä. Noin sadan siirron jälkeen vastustajasi sanoo sinun hävinneen. Vahvistettu oppiminen toimii tällä tavalla. Monissa monimutkaisissa ympäristöissä vahvistettu oppiminen on ainoa mahdollinen tapa opettaa ohjelmaa toimimaan korkealla tasolla. Vahvistetun oppimisen tehtävä on löytää saatujen palkintojen avulla optimaalisin toimintaperiaate ympäristöään varten. (Russel & Norvig 2009, 830-831.) Peliohjelmoinnissa hyvä aloituskohta vahvistettua oppimista varten on Q-oppimisen (Q-learning) algoritmi. Se on helppo toteuttaa ja sitä on testattu laajasti useissa eri sovelluksissa. (Millington, I. 2006. 612-613.)

3.2 Q-oppiminen

Q-oppiminen on algoritmi, jota käytetään vahvistetussa oppimisessa. Sitä kutsutaan mallivapaaksi algoritmiksi, koska se ei yritä luoda kuvaa ympäröivästä pelimaailmasta ja sen lainalaisuuksista. Se kohtelee pelimaailmaa tilakoneena ja on aina jossakin tilassa. Tämän tilan tulee sisältää kaikki oleellinen tieto ympäröivästä pelimaailmasta ja sisäisestä datasta. Jos tila ei sisällä jotain tietoa, sitä ei voida oppia. Jos kahdessa muuten identti-

sessä tilassa on yksikin eroavaisuus (esimerkiksi pelihahmon elämäpisteet), niitä kohdellaan eri tiloina. (Millington, I. 2006. 613.)

Q-learning algoritmi:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$ (Hyrkäs, J. 2015.)

missä $Q(s_t, a_t)$ on toiminto ja tilaparin arvo

s_t on pelin tila

a_t on toiminto

α on oppimisparametri

r on vahvistetun oppimisen arvo

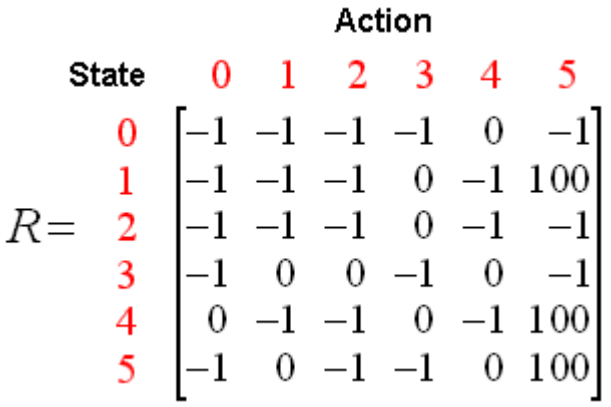
γ on vähennyskerroin

Oppimisparametri α määrittelee, miten paljon nykyisellä arvolla on vaikutusta aikaisemmin tallennettuun arvoon nähden. Jos arvo on nolla, algoritmi ei opi. Jos arvo on yksi, vanhoja arvoja ei käytetä. Vähennyskerroin γ määrittelee miten paljon vaikutusta seuraavilla askeleilla on oppimiseen. Suuri arvo painottaa hyvien lopputilojen löytymiseen ja reitin pituus vaikuttaa siihen vain vähän. Pieni arvo yrittää myös löytää parhaan lopputilan mutta painottaa lyhempään reittiin. (Hyrkäs, J. 2015.)

$Q(s_t, a_t)$ eli toiminnan ja tilaparin arvot voidaan tallentaa matriisiin (kuvio 9). Alussa matriisi on alustettu arvolle 0. Myös vahvistetun oppimisen r arvot voidaan tallentaa matriisiin (kuvio 10). Tämä matriisi sisältää ennalta määrätyt pisteet eri toiminnoille eri tiloissa siirtymien hyödyllisyydestä riippuen. Arvo -1 tarkoittaa, että kyseinen toiminto ei ole mahdollinen kyseisessä tilassa. Kuvion 10 mukaan tilassa 1 toiminto 5 on arvoltaan 100. Vähennyskertoimen γ avulla voidaan säädellä, halutaanko keskittyä enemmän heti tuleviin palkintoihin vai mahdollisiin tulevaisuudessa saataviin pisteisiin. (McCullock, J.)

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Kuvio 9. Q matriisi (McCullock, J.).



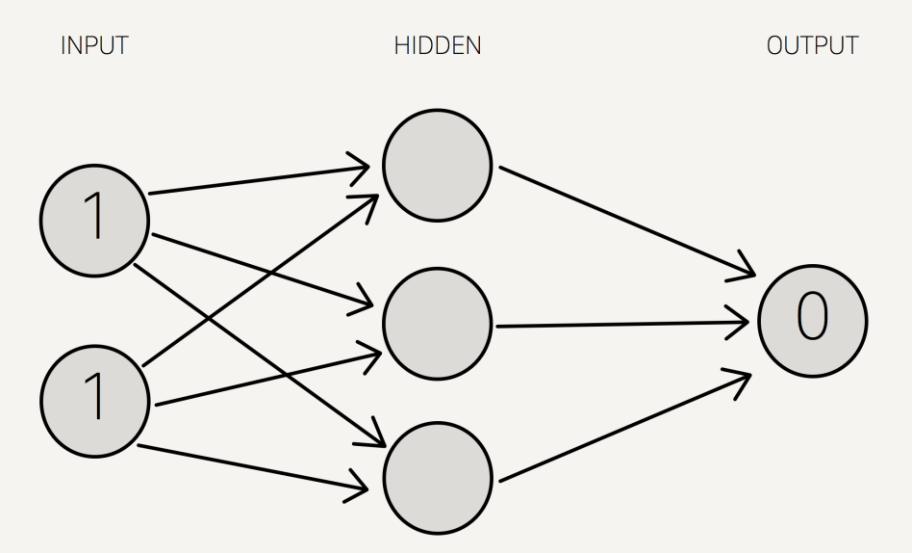
Kuvio 10. R matriisi (McCullock, J.).

Matriisin käyttö on suotuisaa sen helpon toteuttamisen vuoksi, mutta on käytännöllinen vain jos tiloja ja toimintoja on pieni määrä (Dini & Serrano 2012.). Jos tilojen ja toimintojen määrä kasvaa suureksi, on suositeltavaa käyttää neuroverkkoja funktioiden approksimoimiseen (Lin, LJ. 1993.).

3.3 Neuroverkot

Neuroverkko koostuu suuresta määrästä suhteellisen yksinkertaisia solmuja, jotka ajavat samaa algoritmia. Nämä solmut ovat keinotekoisia neuroneja, jotka on alun perin suunniteltu simuloimaan yksittäisen aivosolun toimintaa. (Millington, I. 2006. 628.) Millerin (2015) mukaan neuroverkot ovat ihanteellisia valvottua oppimista varten. Niiden koulutus tapahtuu syöttämällä dataa eteenpäin niiden lävitse, jonka jälkeen ulostulon ja halutun ulostulon välisen erotuksen avulla painotetaan verkon synapsien painoja uudestaan verkkoa taaksepäin.

Kuviossa 11 on esitettynä yksinkertainen neuroverkko, jossa on kaksi syöteneuronia, kolme piilotettua neuronia ja yksi ulostuloneuroni. Neuronien väliset viivat kuvassa ovat synapseja ja ne esittävät neuronien yhteyksiä toisiinsa. Jokaiselle synapsille määritellään aluksi sattumanvarainen paino. Sisään tulleiden arvojen kulkiessa neuroverkon läpi, ne saavat uuden arvon jokaiselle neuronille synapsien painojen perusteella. Laskennassa käytetään aktivointifunktiota, esim. sigmoidin funktiota. Aktivointifunktiot ovat välttämättömiä neuroverkoille, jotka mallintavat epälineaarista käyrää. Ulostulon ja halutun arvon välisen erotuksen avulla lasketaan uudet painot synapseille. Painot päivitetään neuroverkkoa pitkin ulostulosta alkaen sisääntulon suuntaan. (Miller, S. 2015.)



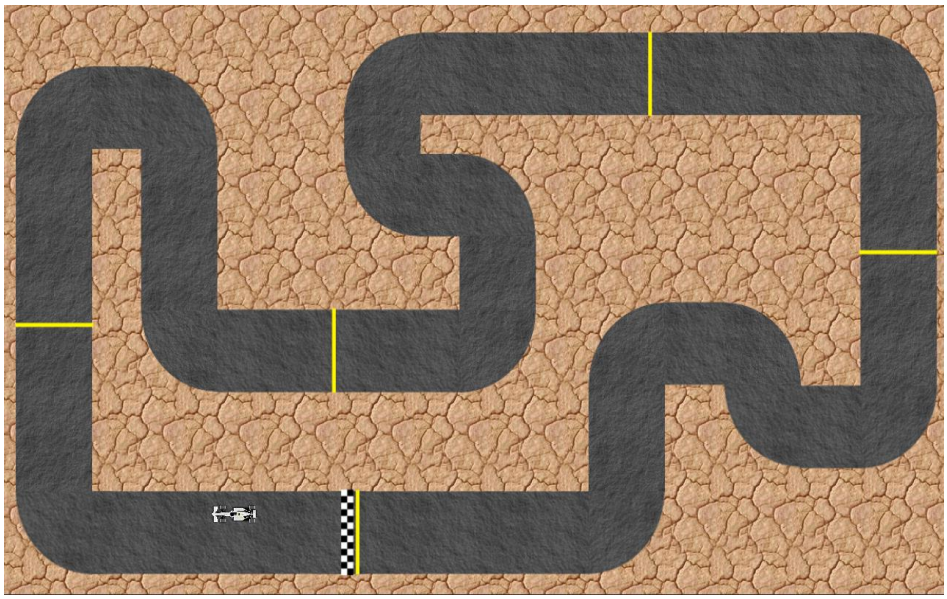
Kuvio 11. Yksinkertainen neuroverkko (Miller, S. 2015).

4 Pelin toteutus

Opiskellessani oppivista tekoälyistä, törmäsin usein Q-oppimisen algoritmiin. Q-oppimisen käytöstä löytyi erilaisia selventäviä esimerkkejä, jotka helpottivat siihen tutustumista. Tarkemman tutustumisen jälkeen päädyin käyttämään Q-oppimista oppivan tekoälyn toteuttamiseen, koska sillä on saatu aikaan hyviä tuloksia niin peleissä kuin robotiikassakin (Sethy, Patel & Padmanabhan 2015, 259). Useissa löytämissäni Q-oppimisen esimerkeissä algoritmin käyttämä data oli tallennettu tauluihin. Hyvin pian huomasin, että pelini tulee sisältämään paljon erilaisia tiloja ja toimintoja, joita Q-oppiminen tarvitsee oppimista varten. Tästä johtuen peliä ei voinut toteuttaa tauluja käyttämällä, vaan oppimiseen tarvittiin neuroverkkoa. Neuroverkko toimii tässä tapauksessa Q-funktiona *Q(st, at).*

4.1 Peli-idean esittely

Ideana on tehdä yksinkertainen ylhäältä kuvattu ajopeli (kuvio 12), jossa kilpaillaan tekoälyä vastaan. Autot liikkuvat itsestään vakionopeudella eteenpäin ja pelaajan ohjattavissa on auton kääntäminen oikealle tai vasemmalle. Ohjaus tapahtuu painamalla näyttöä oikealta tai vasemmalta halutun kääntösuunnan mukaisesti. Nopeuden säätämisen puuttumisen ideana on sekä haasteen lisääminen että tekoälyn mahdollisten toimintojen rajoittaminen. Jos mahdollisten toimintojen määrä kasvaa suureksi, tekoälyn opettaminen monimutkaistuu huomattavasti. Halusin myös pitää pelialueen mahdollisimman yksinkertaisena, jotta tekoäly pystyisi havaitsemaan ympäristöä mahdollisimman selkeästi. Piirsin radan itse Gimpillä (Gimp) ja autojen kuvat ovat Unityn Asset Storesta (Unity Asset Store).



Kuvio 12. QDrive rata ja pelaajan auto.

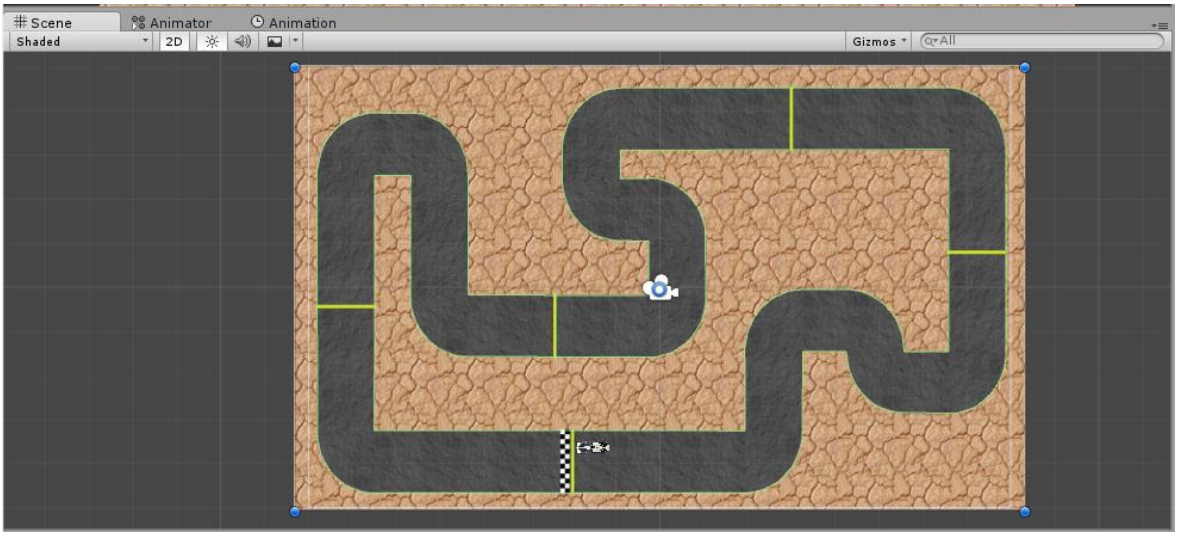
4.2 Pelin tavoite ja suunnittelu

Ideana on luoda aluksi yksinpeli, johon lisätään tekoäly. Koska peli suunnitellaan ihmisen pelattavaksi, tekoälyn täytyy mukautua siihen. Tekoälyn opettaminen tapahtuu antamalla sen opetella itsestään. Mitä kauemmin tekoälyn antaa opetella, sitä paremmin se alkaa ajamaan. Opetus tapahtuu antamalla tekoälyn ajaa rataa ympäri itsestään. Se saa palautetta törmäyksistä ja radalla pysymisestä. Q-oppimisen avulla tekoäly alkaa huomaamaan huonot valinnat ennakkoon, kuten radan reunaa päin kääntämisen. Tavoitteena on oppia ymmärtämään Q-oppimisen toimintaa tekoälyjen ja neuroverkkojen kanssa.

Käytin pelin tekemisessä Unitya, koska se on hyvä ja tunnettu pelimoottori 2D- ja 3D-peleille. Pelimoottorin dokumentaatio on myös hyvä ja kattava. Lisäksi Unity on itselleni ennalta tuttu ja olen käyttänyt sitä omissa projekteissani. Unity on Unity Technologiesin kehittämä pelimoottori, jolla voi kehittää pelejä usealle eri alustalle. Se on henkilökohtaisessa käytössä ilmainen, jos sillä ansaitut vuositulot ovat alle 100 000 dollaria (Unity Terms of Service).

Toteutin Unitylla aluksi yksinpelin. Tällä tavalla tekoälyn täytyy mukautua ihmisen pelattavaksi suunniteltuun peliin. Päätin antaa pelille nimeksi QDrive. Pelin ideana on ajaa rataa ympäri useita kertoja. Jos pelaaja (tai tekoäly) ajaa radalta ulos, auto siirretään takaisin viimeisimmälle ohitetulle tallennuspisteelle (keltaiset viivat, kuvio 12). Ulosajon jälkeen on sekunnin viive, ennen kuin ajoa voi jatkaa. Näiden ominaisuuksien on tarkoitus toimia hidasteena niin pelaajalle kuin tekoälyllekin virheitä tehdessä. Tässä vaiheessa peliin ei ole vielä lisätty tekoälyä.

Kuviossa 13 on Unityn scene-näkymä. Tässä näkymässä voidaan muokata pelissä käytettyjä komponentteja. Esimerkiksi auton sijaintia pystyy vaihtamaan vetämällä sitä hiirellä. Näkymää voi myös käännellä. Rataa reunustavat vihreät viivat ovat Unityn omia törmäyksen tunnistajia, jotka havaitsevat toisten tunnistimien osumat. Autoon on kiinnitetty oma tunnistin. Keltaisten viivojen kohdalla on omat tunnistimet, jotka tallentavat auton paikan niihin osuessa. Radan reunan tunnistimet havaitsevat jos autoissa oleva tunnistin osuu niihin ja siirtää auton takaisin edelliselle ylitetylle keltaiselle viivalle. Tästä seuraa rangaistuksena sekunnin viive, ennen kuin peliä voi jatkaa.



Kuvio 13. Unityn Scene-näkymä.

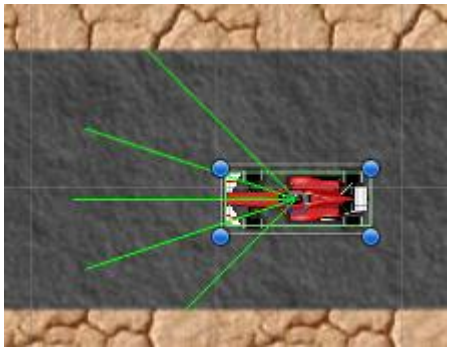
Yhtenä ongelmana oli valita, miten usein algoritmilla valitaan seuraava toiminto. Unity tukee oletuksena päivitystä jokainen ruudunpäivitys tai kiinteästi 20 ms välein. Kiinteää päivitystä suositellaan pelifysiikoissa (Unity Documentation). Kiinteässä laskemisessa on se etu, että se lasketaan, vaikka ruudunpäivitys olisi alhaisempi kuin 20ms:in vaatima 50 ruudunpäivitystä sekunnissa. Päädyin käyttämään tätä kiinteää päivitystä, koska törmäyksen tunnistimet toimivat pelifysiikkojen perusteella. Kaikkien pelissä olevien tunnistimien sijainti tarkistetaan aina 20ms välein ja jos havaitaan törmäys, se käsitellään halutulla tavalla. Tässä pelissä auto siirretään viimeiselle keltaiselle viivalle. Autot eivät voi törmätä toisiinsa, eivätkä tekoälyn sensorit havaitse pelaajaa. Päädyin tähän ratkaisuun, jotta pelaajan tuoma satunnaisuus ei vaikeuttaisi oppimista. Käytännössä tekoäly yrittää vain pysyä radalla, eikä se havaitse pelaajaa ollenkaan.

4.3 Algoritmin suunnittelu ja implementointi

Koska Q-oppimisen algoritmi kohtelee maailmaa tilakoneena, sen täytyy erottaa eri tilat toisistansa. Tämän takia sen täytyy havaita ympäristöään jollain tavalla. Tässä tapauksessa tekoäly havaitsee maailmaa viiden sensorin avulla (vihreät viivat, kuvio 14). Sensorien etäisyys seinistä määrittää tekoälyn tilan. Toimintoja on yhteensä kolme: käänny oikealle, käänny vasemmalle tai mene suoraan. Tilan ja toimintojen yhdistelmä annetaan neuroverkolle. Kyseinen tila täytyy ajaa neuroverkolle kolme kertaa eri toimintojen kanssa (kerran jokaista toimintoa kohden) ja korkein ulostuloarvo määrittää parhaimman toiminnon. Kouluttaessa palkinnoksi tulee kyseisen toiminnon aiheuttama heti saatava palkinto (0.1 pistettä jos ei törmää seinään, 0 jos törmää) ja uuden tilan parhaimman toiminnon pisteet kerrottuna oppimiskertoimella.

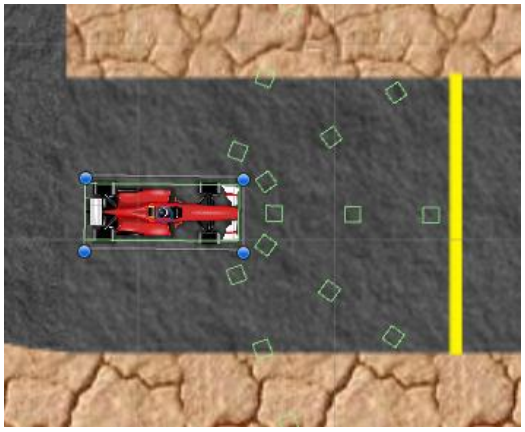
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\max Q(s_{t+1}, a_{t+1})]$$

Q(st, at) on toiminnosta suoraan saatava pistemäärä, α on oppimiskerroin seuraavia tiloja varten ja maxQ(st+1, at+1)] on seuraavista tiloista paras saatava arvo. Jos seuraavasta tilasta paras arvo on 0.12, kerrotaan se α:lla ja lisätään nykyisestä tilasta suoraan saatavaan arvoon. Jokaisesta tilasta, jossa ei törmätä seinään, saa 0.1 pistettä. Joten jos oppimiskerroin on 0.4, tulisi tämän tilan arvoksi: 0.1 + 0.4 * 0.12 = 0.148.



Kuvio 14. Tekoäly ja sensorit.

Tekoälyn toteuttamisen aikana tuli todettua, etteivät taulut ole riittävän tarkkoja tähän tarkoitukseen. Koska tiloja on käytännössä ääretön määrä, ja taulu osaa kertoa toiminnon arvon vain tarkalleen tietylle tilalle, kaikkien tilojen tarkka kouluttaminen on mahdotonta. Neuroverkko sen sijaan osaa kertoa mikä on paras toiminto tämän kaltaisessa tilassa, eikä sen ole täytynyt olla aikaisemmin täsmälleen samassa tilassa osatakseen arvioida parhaan toiminnon. Ennen neuroverkon käyttöönottoa yritin aluksi yksinkertaistaa tekoälyn sensoreita, jotta tilojen määrä laskisi. Kuviossa 15 on nähtävissä yksi näistä kokeiluista. Tekoäly havaitsee, että mitkä sensoreista (15 pientä neliötä) osuvat radan reunaan tai ovat poissa radalta ja määrittää näin oman sijaintinsa suhteessa radan reunoihin. Tässäkin toteutustavassa eri tilojen määrä oli liian suuri ja 15 erillistä sensoria monimutkaistivat opetusta turhaan. Tästä johtuen päädyin käyttämään aikaisempaa viiden sensorin ratkaisua ja neuroverkkoa.



Kuvio 15. Yksinkertaistetut sensorit.

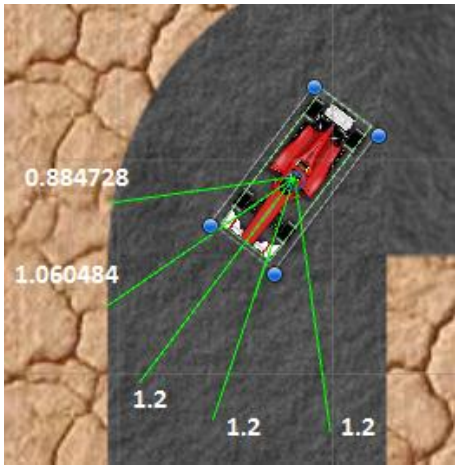
Käytin neuroverkkona löytämääni valmista C#:lla kirjoitettua neuroverkkoa (Schulte, C. 2014), jota muokkasin hieman tarpeeseeni sopivaksi. Neuroverkon toiminnan ymmärtäminen oli tärkeää tuloksia analysoidessa. Luotu neuroverkko sisältää kuusi syöteneuronia (viisi tilan hahmottamiseen ja yksi toiminnolle), seitsemän piilotettua neuronia ja yhden ulostuloneuronin.

Algoritmi ajetaan 20ms:in välein, eli 50 kertaa sekunnissa. Tämä sisältää kaiken kolmen mahdollisen toiminnon ja nykyisen tilan ajamiset neuroverkon lävitse ja ulostulojen analysoinnin. Koska päivityksiä per sekunti on niin paljon ja negatiivista palautetta saa vain törmäyksistä, menee aikaa, että neuroverkko oppii väistämään seinää ajoissa. Aluksi tieto välittyy lähinnä vain 20ms aikaisempaan tilaan. Voisi olla tehokkaampaa tarkistaa tila esim. 100ms välein, jolloin päivityksiä ei olisi niin montaa kuin nyt. Tällaiseen tarkoitukseen 50 päivitystä/sekunti tuntuu aivan liian paljolta. Yksi toiminto voisi myös käsittää esim. 15 asteen käännöksen kerrallaan, nykyisen päivitysväli * kääntymisnopeus sijasta.

4.4 Algoritmin toiminta

Kuviossa 16 tekoäly on opetellut ajamaan 1400 törmäyksen verran ja tallentanut tulokset neuroverkkoon Q-oppimisen avulla. Lähestyessään oikeallaan olevaa seinää, sensorit näyttävät kuvion 16 mukaisia etäisyyksiä. Neuroverkko saa siis seuraavanlaisen listan kolme kertaa (kerran jokaista toimintoa kohti):

new double[] = { 1.2, 1.2, 1.060484, 1.2, 0.884728, ***toiminto*** *(0, 1 tai 2) }*



Kuvio 16. Sensorien etäisyydet.

Eri toimintoja varten on määritelty numeroarvot, jotta ne voi syöttää neuroverkolle. 0 on suoraan, 1 on oikealle ja 2 on vasemmalle. Kun etäisyydet syötetään neuroverkolle kaikkien toimintojen kanssa, ulostulot ovat seuraavaa:

OUTPUTS: forward: 0.150231143105206, right: 0.165422217807077, left: 0.168124554621198
Chosen action: left

Näistä valitaan suurin arvo, joka on tässä tapauksessa vasen (left). Kuviossa 16 tekoäly on siis kääntymässä vasemmalle ja välttää näin seinän.

4.5 Lopputulos

Tekoäly ei opi aivan joka kerta ajamaan rataa oikein. Tämä saattaa johtua mahdollisista painotus- tai opetusongelmista neuroverkossa. Tarkempi ongelman selvittäminen vaatisi jatkotutkimuksia. Tekoälyn oppimista voisi tehostaa ja liikkumista sulavoittaa erilaisilla säädöillä ja optimoinneilla, mutta tämän opinnäytetyön puitteissa aika ei riitä siihen. Myös haluttu vaikeusasteen säätö vaatisi erilaisia optimointeja, sillä nykyisellään tekoäly joko osaa ajaa rataa lähes täydellisesti tai ei lainkaan. Yhtenä optimointina voisi palkita tekoälyä kun se ajaa mutkissa lähellä sisäkaarta, jolloin rata-aikoja voitaisiin saada paremmiksi. Tällä hetkellä tekoäly vain välttelee seiniä.

Tekoäly oppii useimmiten ajamaan rataa lähes täydellisesti viimeistään noin 2000 törmäyksen jälkeen. Yhden päivityksen pituus on niin lyhyt, että tekoälyllä menee aikaa, jotta tieto törmäyksestä osataan arvioida tarpeeksi ajoissa. Tutkimuksen perusteella tekoäly osaa opittuaan arvioida tilanteet oikein ja pyrkii välttämään törmäyksiä toiminnoillaan. Q-oppiminen tuntuu toimivan hyvin ja tieto välittyy verkossa halutusti.

5 Yhteenveto

Opinnäytetyöni muuttui hieman alkuperäisestä suunnitelmastani teon aikana. Alkuun tuli ongelmia löytää esimerkkejä oppivista tekoälyistä, koska niitä ei käytetäkään niin usein kuin olin ymmärtänyt. Yleensä myös niiden opetus toteutetaan studiolla eikä pelaamisen aikana, joten esimerkkejä omaa projektiani varten oli vaikea löytää. Toiseksi ongelmaksi tuli taulujen käyttö Q-oppimisen kanssa. Mietin aluksi yksinkertaisempaa ratkaisua tekoälyn sensoreita varten, jotta tiloja olisi vähemmän. Tilojen määrää ei kumminkaan pystynyt rajoittamaan paljoa, jottei auton radan tunnistus heikkenisi. Koska taulut osoittautuivat liian hitaaksi vaihtoehdoksi, täytyi opinnäytetyöhön lisätä neuroverkko. Olisi ollut mielenkiintoista tehdä neuroverkko alusta asti itse, mutta ikävä kyllä opinnäytetyöhön varattu aika ei olisi millään riittänyt siihen. Sain tutustuttua useampaan erilaiseen toteutukseen, joista valitsin tätä työtä vartein selkeimmän. Neuroverkko toi mukanaan implementaatioongelmia ja vaikeutti työtä alkuperäisestä suunnitelmasta huomattavasti. Neuroverkon hienosäätöön voisi käyttää paljon aikaa, mutta tämän työn puitteissa siihen ei ollut resursseja. Lopputulos oli kuitenkin positiivinen. Tekoäly oppii ajamaan itse Q-oppimisen avulla. Jos peli olisi yksinkertaisempi, neuroverkon voisi korvata tauluilla. Tällöin tekoälyn toteuttamisesta tulisi huomattavasti helpompaa. Q-oppimisen toiminnan pystyisi arvioimaan jo yksinkertaisemmallakin pelillä.

Tutkimukseni tarkoituksena oli selvittää miten oppiva tekoäly lisätään mobiilipeliin ja onko se kannattavaa sekä pohtia mahdollisia ongelmia. Eniten ongelmia tuotti oppivan tekoälyn suunnittelu ja toteutus, mutta sen lisääminen mobiilipeliin ei tuonut mukanaan lisähaasteita. Oletin myös neuroverkon tuovan mahdollisia suorituskykyongelmia, mutta oma matkapuhelimeni jaksoi pyörittää peliä sujuvasti. Toki olisi tarpeen tutkia pelin toimintaa erilaisilla puhelimilla, mutta laaja testaus ei kuulunut opinnäytetyön tavoitteisiin. Oppivan tekoälyn kannattavuudesta tulin siihen tulokseen, että sillä saa ainakin tämän projektin kaltaisissa peleissä luotua hyvän pelikohtaisen tekoälyn. Mutta pelikohtaisesti arvioisin erikseen, onko oppiva tekoälyn lisääminen sen tuoman lisävaivan arvoista. Vaikka tekoäly ei opi pelaamisen aikana, kuten alun perin oli tarkoitus, tekoälyvastustajan sai koulutettua hyväksi kuskiksi, joka ajaa rataa siististi. Oppivan tekoälyn saa siis opetettua pelikohtaisesti pelaamaan hyvin, mutta itse oppivuus pelaamisen aikana tuottaisi haasteita. Yleisen käytännön mukaisesti (Millington, I. 2006.) päätyisin itse opettamaan tekoälyä ennen pelin julkaisua ja jättämään tekoälyn haluttuun tilaan lopulliseen peliin. Yksinkertaisia pelejä varten löytyy todennäköisesti helpompiakin tapoja toteuttaa tekoäly, ellei peliin välttämättä tarvita oppivaa komponenttia.

Sain tämän projektin aikana hyvän kuvan Q-oppimisen ja neuroverkon yhdistämisestä.

Myös tekoälyjen käyttö peleissä tuli hyvin tutuksi. Aikomuksenani on käyttää näitä tietoja tulevaisuudessa omissa projekteissani. Toivon, että näistä tuloksista voivat hyötyä myös muut pelinkehittäjät.

Lähteet

Bilby. 2008. Turing Test. https://www.learner.org/courses/physics/visual/visual.html?shortname=turing_model . Luettu: 2.3.2016.

Copeland, J. 2000. Is Strong AI Possible? Luettavissa: http://www.alanturing.net/turing_archive/pages/reference%20articles/what_is_AI/What%20is%20AI13.html. Luettu 28.3.2016.

Dini, S & Serrano, M. 2012. Combining Q-Learning with Artificial Neural Networks in an Adaptive Light Seeking Robot. Luettavissa: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.297.6355&rep=rep1&type=pdf. Luettu: 22.5.2016.

Eranki, R. 2002. Pathfinding using A* (A-Star). Luettavissa: http://web.mit.edu/eranki/www/tutorials/search/. Luettu 28.3.2016.

Giantbomb. Artificial Intelligence. Luettavissa: http://www.giantbomb.com/artificial-intelligence/3015-218/. Luettu 28.3.2016.

Gimp. https://www.gimp.org/.

Hietala, J. 2012. Tekoälyn sovellutukset peleissä. Case Study: DroidWars. Luettavissa: http://urn.fi/URN:NBN:fi:amk-2012121219185. Luettu 28.3.2016.

Hyrkäs, J. 2015. Reinforcement learning in a turn-based strategy game. Luettavissa: http://urn.fi/URN:NBN:fi:amk-2015120419354. Luettu 22.5.2016.

Kahanda, I. 2011. Difference Between Strong AI and Weak AI. Luettavissa: http://www.differencebetween.com/difference-between-strong-ai-and-vs-weak-ai/. Luettu 28.3.2016.

Kehoe, D. 2009. Designing Artificial Intelligence for Games (Part 1). Luettavissa: https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1. Luettu 28.3.2016.

LaCurts, K. Criticisms of the Turing Test and Why You Should Ignore (Most of) Them. Luettavissa: people.csail.mit.edu/katrina/papers/6893.pdf. Luettu: 4.12.2016.

Lee, R. 2015. AI Learns Chess In Just Three Days, Now Plays Equivalent To International Grandmaster Level. Luettavissa: http://www.techtimes.com/articles/86476/20150921/ai-learns-chess-in-just-three-days-now-plays-equivalent-to-international-grandmaster-level.htm. Luettu 30.3.2016.

Lewis, T. 2015. Google's Artificial Intelligence Can Probably Beat You at Video Games. Luettavissa: http://www.livescience.com/49947-google-ai-plays-videogames.html. Luettu 28.3.2016.

Lin, LJ. 1993. Reinforcement Learning for Robots using Neural Networks. Luettavissa: www.dtic.mil/dtic/tr/fulltext/u2/a261434.pdf. Luettu: 22.5.2016.

McCullock, J. A Painless Q-Learning Tutorial. Luettavissa: http://mnemstudio.org/path-finding-q-learning-tutorial.htm. Luettu: 15.5.2016.

Microsoft b. Video Games and Artificial Intelligence. Luettavissa: http://research.microsoft.com/en-us/projects/ijcaiigames/ijcaiislides1.pptx. Luettu 28.3.2016.

Miller, S. 2015. Mind: How to Build a Neural Network (Part One). Luettavissa: http://stevenmiller888.github.io/mind-how-to-build-a-neural-network/. Luettu: 2.10.2016.

Millington, I. 2006. Artificial Intelligence for games. Morgan Kaufmann Publishers. Luettavissa: http://ldc.usb.ve/~vtheok/cursos/ci5325/lecturas/Artificial%20Intelligence%20for%20Games.pdf. Luettu: 15.5.2016.

Patel, A. Introduction to A*. Luettavissa: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#dijkstras-algorithm-and-best-first-search. Luettu: 30.3.2016.

Pearce, A. 2016. Top 10 video games of the 70s. Nerdblock. Luettavissa: https://www.nerdblock.com/blog/video-games-of-the-70s/. Luettu: 20.11.2016.

Poole, D. & Mackworth, A. 2010. Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press. Luettavissa: http://artint.info/html/ArtInt.html. Luettu: 15.5.2016.

Prado, G. 2015. Intelligent robots don't need to be conscious to turn against us. Luettavissa: http://www.techinsider.io/artificial-intelligence-machine-consciousness-expert-stuart-russell-future-ai-2015-7. Luettu: 2.4.2016.

Programmer Interview. What's the difference between DFS and BFS? Luettavissa: http://www.programmerinterview.com/index.php/data-structures/dfs-vs-bfs/. Luettu: 30.4.2016.

Russel, S. & Norvig, P. 2009. Artificial Intelligence: A Modern Appriach (3rd Edition). Luettavissa: http://cessa.khu.ac.ir/wp-content/uploads/2015/12/Artificial-Intelligence-A-Modern-Approach-3rd-Edition.pdf. Luettu: 11.12.2016.

Schulte, C. 2014. Simple C# Artificial Neural Network. Luettavissa: http://www.craigsprogramming.com/2014/01/simple-c-artificial-neural-network.html. Luettu: 1.4.2016.

Sethy, H. & Patel A. & Padmanabhan, V. 2015. Real Time Strategy Games: A Reinforcement Learning Approach. Luettavissa: http://www.sciencedirect.com/science/article/pii/S187705091501354X. Luettu: 11.12.2016.

Unity Asset Store. Race Cars 2D. https://www.assetstore.unity3d.com/en/#!/content/18207.

Unity Documentation. MonoBehaviour.FixedUpdate(). Luettavissa: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html. Luettu: 12.12.2016.

Unity Terms of Service. Luettu: 2.12.2016. Luettavissa: https://unity3d.com/legal/terms-of-service/software.

Wexler, J. 2002. Artificial Intelligence in Games: A look at the smarts behind Lionhead Studio's "Black and White" and where it can and will go in the future. Luettavissa: http://www.cs.rochester.edu/~brown/242/assts/termprojs/games.pdf. Luettu: 28.3.2016.